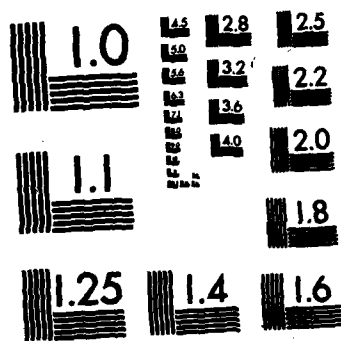END

FILMED

DTIC

MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

MAN-MACHINE COOPERATION FOR ACTION PLANNING

Final Report

November 1982

By: Ann Robinson, Senior Computer Scientist
David Wilkins, Computer Scientist

Artificial Intelligence Center
Computer Science and Technology Division

DTIC
S D
FEB 9 1983
A

MAN-MACHINE COOPERATION FOR ACTION PLANNING

Final Report

November 1982

By: Ann Robinson, Senior Computer Scientist[*]
David Wilkins, Computer Scientist

Artificial Intelligence Center
Computer Science and Technology Division

1. Introduction

This is the final report for SRI Project 1349, Office of Naval Research Contract N00014-80-C-0300, which investigated the cooperative process that enables a computer to assist a decisionmaker in planning and scheduling sequences of actions. This involved the development of a new system for planning and scheduling actions, along with a human-engineered package for defining multimodal man-machine interfaces (i.e., interactions using different human senses) that can be readily intermingled. In addition to work on these two aspects of the general problem, we produced a demonstration system applying the techniques devised in the course of the project to a task of relevance to the Navy.

As a representative application, we selected the problem of planning and monitoring aircraft movement on board a carrier. This choice was made with the aid of Captain Richard L. Martin, his officers, and crew from the USS Carl Vinson, who have provided invaluable assistance in identifying problems associated with aircraft handling.

2. SPOT – Planning and Monitoring Plane Movement

The experimental system developed by us, SPOT, was designed to assist in planning the movement and launching of planes on a carrier. SPOT ([1],[2]) was built by extending SIPE, a general-purpose system for planning developed at SRI International.[1] SIPE ([3]) supports domain-independent planning and provides the capabilities needed for hierarchical planning, parallel actions, and monitoring of plan execution. SPOT can generate plans automatically, but, unlike its predecessors, it is designed to also allow interaction with users throughout the planning and plan execution processes, if so desired. The user is able to watch and, when desired, guide and/or control the planning process.

An example of a problem given to SPOT is to plan the first two launch cycles of the day and the first recovery. Planning a launch entails determining the order for launching planes,

---

1

ensuring that they are "up," fueled, and have crews, deciding which catapults to launch them from, and ensuring an unobstructed path to the catapult.

SPOT contains information about the aircraft, such as their status, readiness, and location. It also contains information about actions associated with moving and launching aircraft, the requirements for performing these actions, their anticipated effects, and how they can be carried out. For example, the action of moving a plane to a catapult requires a clear path (one might have to be cleared), its result is that the plane is in a new location, and it is carried out by actually moving the plane.

Planning in SIPE and SPOT is performed hierarchically: from general to specific actions. In planning a launch cycle, the general launch sequence is planned first, then the details for each plane. This is like planning the major components of a house before one starts worrying about each individual board and nail.

A plan can contain actions to be performed sequentially or in parallel. SPOT can explore alternative sequences of actions, e.g., launching planes in another order. SPOT can also ascertain the actions that could be taken in a specific situation and allow exploration of several alternatives. For example, it can determine different ways of preparing a plane for launch and make it possible for the user to develop plans that incorporate the various options in turn.

3. Sample Problem Solution

This section traces through the steps performed by SPOT while planning a day's activity on board the carrier.

The highest level of description of two launch cycles and a recovery might be represented by the three steps: Cycle1, Cycle2, Recovery1. Cycle1 and Cycle2 are both sets of launches of several planes in parallel.

SPOT first expands Cycle1 to the set of parallel launches, then orders them. In this problem it places the SH-3s first, then the E-2C, then the S-3, and finally the A-7 and the KA-6 (the

latter two were left in parallel).

SPOT then expands the launches one more level, which involves assignment of the individual planes. This is done first for the SH-3s by producing a four-step launch plan for each helicopter. A launch plan consists of (1) satisfying the goal that the aircraft is NOT DOWN, (2) satisfying the goal that the aircraft is READY, (3) satisfying the goal that the aircraft is AT the launch location, and (4) performing the TAKEOFF. Two helicopters are then selected (either manually or by the system) so that the NOT DOWN goal is satisfied, if possible.

The READY goal must be planned to another level in the hierarchy, since it requires that the aircraft be fueled and armed, and a suitable crew found. SPOT actually provides two actions for making an aircraft READY: fast-ready makes sure it is partly fueled and has a crew, while full-ready fuels it completely, arms it, and finds a crew.

Satisfying the AT goal involves finding a clear path from the aircraft's current location to the launching catapult. This requires planning at a lower level and makes use of the MOVE operator for moving an aircraft from one location to another (possibly to clear a path). The TAKEOFF action is primitive.

SPOT proceeds through each launch in Cycle1 and Cycle2, planning down through all these levels, until all the goals have been accomplished by primitive actions.

4. Interaction with the User

One of the key features of SPOT is its ability to interact gracefully with the user. This is done primarily through a color display. Figure 1 shows a typical layout of the deck of a Nimitz-class carrier as presented by the program. SPOT shows both this information on the screen, along with the plan (or plans) being constructed, and (when desired) status information. As aircraft movement is planned, the display is updated to show the expected deck configuration, based on the model of the domain maintained by SIPE. Although the routines in SPOT display a carrier deck and planes, they have been designed to enable easy encoding of other domains as

well. SPOT also shows the plan being constructed. The user can choose to see different parts of the plan in as much detail as desired.

The user can easily invoke planning operations at any level without being required to make tedious choices that could be performed automatically. SPOT provides a graphic interface to these actions. The user can direct low-level and specific planning operations (e.g., "instantiate PLANE1 to N2636G", "expand CYCLE1 with the LAUNCH operator"), high-level operations that combine the lower-level ones (e.g., "expand the whole plan one more level and correct any problems"), or operations at any level between the two (e.g., "allocate resources", "expand CYCLE1 with any operator", "find and correct harmful interactions"). The examples above need not be given as text, since the interactive component of SPOT makes use of graphics and has been implemented on a high-resolution black-and-white bit map display and a color graphics terminal. The planning choices available to the user appear in a menu from which one can be selected by pointing with a mouse or joystick. Similarly, steps in a plan (nodes in the network) can be referred to either by name or by pointing to them.

When decisions have been made by the user, SPOT prevents the creation of any plans that would lead to problems, such as blocking a plane that is to be moved at some later time. Figure 2 is a picture of the display taken during an actual planning session. The current deck configuration of the carrier is shown at the top. In the lower left is the menu of planning choices for the user, and in the lower right is the current plan displayed as a graph similar to a PERT-chart.

5. Details of the Representation

As we have mentioned, SPOT has been developed by using the general-purpose planning system SIPE. Actions planned by SPOT (such as LAUNCH, MOVE, and FUEL) are encoded as operators in SIPE. Planes, catapults, and crews are designated as resources associated with such actions. SIPE's resource allocation capability [3] performs the task of reasoning about these

4
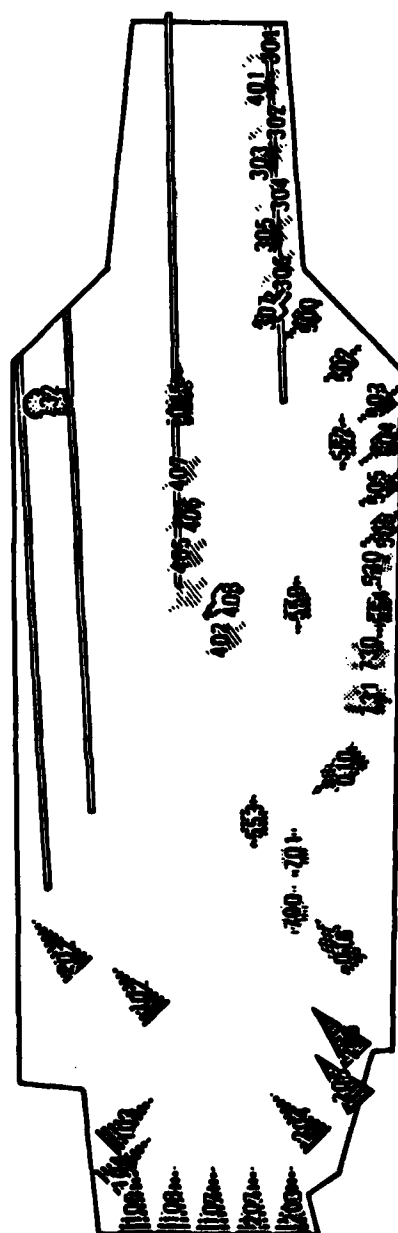
Figure 1

SPOT's display of carrier deck

5
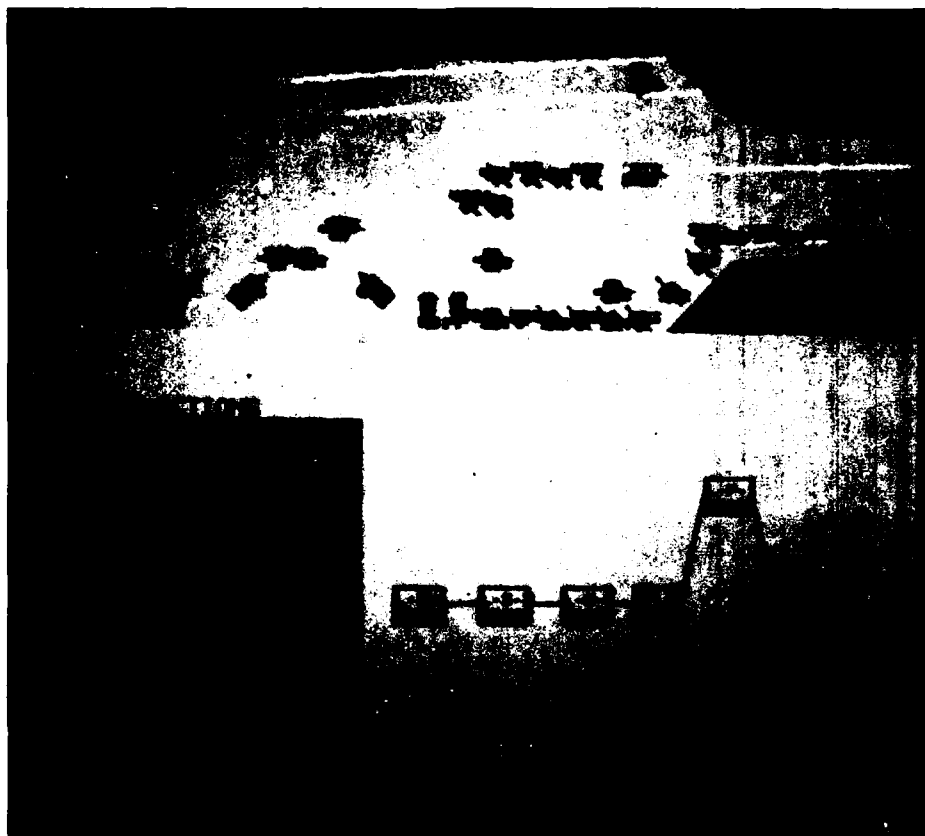
Figure 2

SPOT's display of carrier deck, planning choices, and plan

resources. The preconditions and effects of actions are used to ensure that proposed actions can be performed and to help detect problems during plan execution.

Figure 3 shows part of the move operator used in SPOT. The move operator is designed to plan the movement of OBJECT1 from LOC1 to LOC2, following PATH1. ITERATE1 is a special kind of variable in SIPE. In this operator, its value ranges over PATH1. When this move operator is used to plan a move action, the matching routine that identifies the resources and arguments will recognize the special variable and call a prespecified routine that will generate a

6

```
OPERATOR: MOVE
RESOURCES: OBJECT1, OBJECT2 CLASS UNIVERSAL IS NOT OBJECT1;
ARGUMENTS: LOC2, LOC1,
    PATH1 WITH ORIGIN LOC1 WITH DESTINATION LOC2,
    ITERATE1 WITH RANGE PATH1;
PRECONDITIONS: (AT OBJECT1 LOC1);
PURPOSE: (AT OBJECT1 LOC2);
PLOT:
    PARALLEL-EXPAND
            GOALS: (NOT (AT OBJECT2 ITERATE1));
    END PARALLEL-EXPAND

    PROCESS
    ACTION: MOVE.PRIMITIVE;
    RESOURCES: OBJECT1;
    EFFECTS: (MOVED OBJECT1 LOC1 LOC2 PATH1);
END PLOT
END OPERATOR
```

Figure 3
A MOVE operator in SPOT

list of locations along the path. In addition, the PARALLEL-EXPAND step in the PLOT will expand into a parallel set of goals of the form (NOT (AT OBJECT2 ITERATE1)), one for each value of ITERATE1 along PATH1. This checks for the presence of any object along the path. If no object is at ITERATE1, a *phantom* goal node (i.e., a goal state that is already true) will be created in the plan for monitoring purposes (see below). If there is an object at ITERATE1, SPOT or the user can plan to move it. MOVE.PRIM is the plan step for actually moving the object along the path.

This move operator also illustrates one of the extensions added to SIPE during the development of SPOT: the ability to generate arguments when predicates are being matched. This was introduced to facilitate path planning, principally because algorithms for finding optimal paths were not easily encoded within the existing formalism. The extension of SIPE is not limited in its application to a particular problem, but provides a general mechanism for generating variable values.

7

The representation used in SPOT is described in greater detail in [3] which is included as Appendix of this report.

6. Monitoring a Plan's Execution

SPOT also monitors execution of the plan and, in some circumstances, can detect existing or potential problems during execution. As actions are performed or situations change, they are reported to the system. SPOT checks to determine whether any reported change affects a planned action.

For example, a launch requires that the plane be UP. This is encoded as a precondition of the action. If a plane's launch is planned on the assumption that the plane is UP and then it is reported as DOWN, SPOT will detect that the launch action cannot be carried out as planned; it will then suggest such alternatives as repairing the plane that is down, launching a different plane, or revising the launch plan, - perhaps with fewer F-14s. The user must decide which option to accept. For example, if his decision is to launch another F-14, the computer will show what is available and, when one has been selected, the plan and display will be updated.

Paths for planes are kept clear by a similar process. As discussed above, for each plane's path there are either actions in the plan to clear that path or phantom goal nodes associated with locations that are to remain clear. If something is moved that blocks a plane's path (and that movement is reported to SPOT), the phantom goals that are no longer true will be identified and the handler will be notified of the problem. SPOT will suggest modifications to correct the problem, such as moving the blocking equipment elsewhere, clearing another path for the plane, or selecting another aircraft. Such blocking might occur during execution of a plan that includes the launch of a plane located on the hangar deck. The plan ensures that there will be a clear path through the elevator; however, as the launch starts, another plane may go down and need to be moved to the elevator, blocking the path of the plane from the hangar deck. The computer will detect that this causes a problem and it will notify the user

8

and give him the available options.

SPOT is one example of the use of artificial intelligence techniques for planning. There are many other problems for which similar systems could be built – for example, planning and monitoring a large construction task or the movement of people and equipment. In each of these, an interactive computer system like SPOT could enhance substantially the decision-making capabilities of any individual planner or planning organization.

REFERENCES

1. Robinson, A.E., and Wilkins D.E., "Representing Knowledge in an Interactive Planner" *Proceedings of the First Annual Conference of the AAAI*, Stanford, California, 1980, pp. 148–150.

2. Wilkins, D.E., and Robinson, A.E., "An Interactive Planning System", Artificial Intelligence Center Technical Note 245, SRI International, Menlo Park, California, July 1981.

3. Wilkins, D.E., "Domain Independent Planning: Representation and Plan Generation", Artificial Intelligence Center Technical Note 266, SRI International, Menlo Park, California, August 1982.

APPENDIX

DOMAIN INDEPENDENT PLANNING:
REPRESENTATION AND PLAN GENERATION

Technical Note No. 266

August 1982

By: David Wilkins, Computer Scientist
Artificial Intelligence Center
Computer Science and Technology Division

ABSTRACT

A domain independent planning program that supports both automatic and interactive generation of hierarchical, partially ordered plans is described. An improved formalism for representing domains and actions is presented. The formalism makes extensive use of *constraints*, provides efficient methods for representing properties of objects that do not change over time, allows specification of *purposes* for determining plan rationale, allows specification of *resources*, and provides the ability to express deductive rules about how the world works. The system deduces the effects of actions using deductive rules. The implications of allowing parallel actions in a plan or problem solution are discussed. New techniques for efficiently detecting and remedying harmful parallel interactions are presented. The most important of these techniques, reasoning about resources, is emphasized and explained. The system supports concurrent exploration of different branches in the search, making best-first search easy to implement. Meta-planning and its implications for domain independent planning are discussed in some detail, and primitive execution monitoring capabilites are described.

1

1. The Planning Problem

The problem of generating a sequence of actions to accomplish a goal is referred to as *planning*. To automate planning in a computer program involves representing the world, representing actions and their effects on the world, reasoning about the effects of sequences of such actions, reasoning about the interaction of actions that are taking place concurrently, and controlling the search so that plans can be found with reasonable efficiency.

The ability to reason about actions is a core problem for artificial intelligence. It is part of the common-sense reasoning people do all the time. By reasoning about actions, a program could plan your travel for you, control a robot arm in a changing environment, get a computer system or network to accomplish your computing goals, or any number of other things that are beyond the scope of current systems. This problem is even central to conversing in natural language. First, because many utterances are deliberately *planned* to achieve specific goals, and also because it is often necessary to create your own model of other people's plans in order to understand their utterances. Despite the importance of the problem, relatively little has been accomplished in recent years. This paper describes progress on this problem that has resulted from development of a planning program at SRI International.[1]

Planners designed to work efficiently in a single problem domain, though desirable, often depend on the structure of that domain to such an extent that the underlying ideas cannot be readily used in other domains. Currently, a new program (often requiring its own unique representation and heuristics) must be developed for each new domain (if the domain is complex enough to be interesting) in order to get reasonable performance. This paper discusses domain independent planners which are of particular interest since they provide planning techniques that are applicable in many domains, and provide a general planning capability. Such a common-sense planning ability is likely to involve different techniques than those used by an expert planning in his particular domain of expertise, but is a necessary ability for people in

their daily lives and for intelligent programs. Of course, a general planner should provide representations and methods for including domain specific knowledge and heuristics.

There is no guarantee, except for human performance, that a large central core of domain independent planning techniques exists. It is important to enlarge this core as much as possible so we do not have to write a new planner for each new domain. Such motivation lies behind other AI research; for example EMYCIN [15] is an attempt to clarify the domain independent core of expert systems such as MYCIN, and TEAM [3] provides natural language access to data bases independent of the domain or structure of the data base. This paper describes an implemented planning program that expands the core of domain independent planning techniques as it builds on and extends previous domain independent planning systems such as Sacerdoti's NOAH [10], Tate's NONLIN [14], Sridharan's PLANX10 [11], Vere's DEVISER [16], and SRI's STRIPS [1].

Two features found in many planning systems are also central in this work: hierarchical planning, and parallel actions. Hierarchical planning is often necessary for real-world domains since it helps avoid the tyranny of detail that would result from planning at the most primitive level. By first planning at more abstract levels, the planner can reduce the search space by expanding these more abstract plans into more detailed plans. Parallel actions are also useful for real-world domains. Such domains are often multi-effector or multi-agent (e.g., having two robot arms to construct an object, or having two editors to work on your report), and the best plans should use these agents in parallel when possible. This distinguishes planning from much of the work in program synthesis, since the goal there is often a strictly sequential program. A planning system that allows parallel actions must be able to reason about how actions interact with each other since interference between parallel actions may prevent the plan from accomplishing its goal. This is a major problem for planning systems and a major focus of this paper.

3

2. Overview of SIPE

We have designed and implemented (in INTERLISP) a system, SIPE, (System for Interactive Planning and Execution monitoring), that supports domain independent planning. The program has produced correct parallel plans for problems in four different domains (the blocks world, cooking, aircraft operations, and a simple robotics assembly task). The system allows for hierarchical planning and parallel actions. Development of the basic planning system has led to several extensions of previous systems. These include the development of a perspicuous formalism for encoding descriptions of actions, the use of constraints to partially describe objects, the creation of mechanisms that permit concurrent exploration of alternative plans, the incorporation of heuristics for reasoning about resources, mechanisms that make it possible to perform simple deductions, and advanced abilities to reason about the interaction between parallel actions.

SIPE can automatically generate plans, but, unlike its predecessors, SIPE is designed to also allow interaction with users throughout the planning and plan execution processes, if this is desired. The user is able to watch and, when desired, guide and/or control the planning process. Our concern with interaction means that perspicuous representations have been favored and that some search control issues have not been addressed as yet. As the system evolves, more methods for controlling the search will be developed and more problems will be solved automatically. This evolutionary approach has several advantages. From the planning point of view it allows us to address larger, "real-world" problems which may initially be beyond the capabilities of fully automatic planning techniques, but which could provide interesting research problems. Development of an interactive planner also encourages us to deal with the issue of representing the planning problem in terms that can be easily communicated to a user. This is also important for an automatic planner, because the machine must still be able to communicate about the planning it has performed. Our system raises issues in human-machine interaction, but this paper addresses only the planning aspects of our work.

In SIPE, a plan is a set of partially ordered goals and actions, which is composed by the

4

system from operators (the system's description of actions that it may perform). Plans that do not achieve the desired goal may sometimes be generated by simply applying operators, so the system also has *critics* which find possible problems and attempt to correct them. In particular, most of the reasoning about interactions between parallel actions is done by the critics. The plans are represented in procedural nets [10], primarily for graceful interaction between man and machine. Invariant properties of objects in the domain are represented in a type hierarchy, which allows inheritance of properties and the posting of constraints on the values of attributes of these objects. The relationships which change over time, and therefore all goals, are represented in a version of first-order predicate calculus which is typed and interacts with the knowledge in the type hierarchy. Operators are represented in an easily understood formalism we have developed [7] in which the ability to post constraints on variables is a primary feature. Each of these parts of the system will be described in more detail later in the paper.

It should be noted here that, like most domain independent planning systems (DEVISER being an exception), ours assumes discrete time, discrete states, and discrete operators. Time need not be represented explicitly for many tasks since the ordering links in the procedural network provide the necessary ordering information. It is assumed that each world state that can be reached is discrete and can be represented explicitly. The operators are also discrete, with the effects of an action occurring instantaneously as far as the system is concerned. This applies to a given abstraction level; using hierarchical planning, the system can order the effects which occur at a lower level of detail. These assumptions are acceptable in many real-world domains and have been made by most previous planners. They are however restrictive and prevent many real-world phenomena from being adequately represented. For example, sophisticated reasoning about time and modelling of dynamic processes are not possible within our present framework. Few artificial intelligence programs have addressed these problems, McDermott's recent work being a notable exception [6].

This paper describes SIPE in more detail by giving its solutions to four major problems a

5

planner must address. These problems are representation (of the domain, goals, and operators), recognizing and dealing with parallel interactions, controlling the search, and monitoring execution. The next four sections describe these four problems, and stress new developments in SIPE by comparing it to previous systems. Most of the new developments are in the first two areas; the latter two being areas of some progress in which research is continuing.

3. Representation

One of the central concerns in designing a representation for a planning system is how to represent the effects an action has on the state of the world. This means the frame problem [10] must be solved in an efficient manner. Since many domains will hopefully be encoded in the planning system, it is also necessary that the solution to the frame problem not be too cumbersome. For example, one does not want to have to write a large number of frame axioms for each new action that is defined.

The planning representation problem involves representing the domain, goals, and operators. Operators are the system's representation of actions that may be performed in the domain or, in the hierarchical case, abstractions of actions that can be performed in the domain. An operator includes a description of how each action changes the state of the world. In a logical formalism such as Rosenschein's adaptation of dynamic logic to planning [9], the same representational formalism may be used for representing the domain, goals, and operators, but in many planners more concerned with efficiency, including SIPE, there are different representations for each. The goal is to have a rich enough representation so that many interesting domains can be represented (an advantage of logical formalisms), but this must be traded off against the ability of the system to deal with its representations efficiently during the planning process.

6

3.1 Representation of domain and goals

The system provides for representation of domain objects and their invariant properties with nodes linked in a hierarchy. Invariant properties do not change as actions planned by the system are performed (e.g., the size of a frying pan does not change when you cook something in it). Each node can have attributes associated with it, and can inherit properties from other nodes in the hierarchy. The values of attributes may be numbers, pointers to other nodes, key words the system recognizes, or any arbitrary string (which can only be used by checking if it is equal to another such string). The attributes are an integral part of the system, since planning variables are also nodes in the hierarchy and they contain constraints on the values of attributes of possible instantiations. Constraints are an important part of the system and are discussed in considerable detail later. There are different node types for representing variables, objects, and classes, but these will not be discussed in detail here since they are similar to those occurring in many representation formalisms; for example, semantic networks and UNITS [12].

All parts of the system are actually nodes in this hierarchy; for example, procedural net nodes, operators, and goals are all represented as nodes in this hierarchy. However, unlike variables and domain objects, these latter nodes are near the top of the hierarchy and do not make use of the inheritance of properties, nor the constraints on attribute values. The use of a uniform formalism for all parts of the system has been helpful both in implementation and in interaction with users.

Properties of domain objects and relationships among them that may change as actions are performed are represented in first-order logic. Thus, logic is used to describe goals as well as the preconditions and effects of operators. Quantifiers are allowed whenever they can be efficiently handled. Universal quantifiers are always permitted, existential quantifiers can occur in the preconditions of operators but not in the effects. Disjunction is not allowed. These restrictions result from using "add lists" to solve the frame problem. (Why this is so is described in the next section). By separating the invariant properties of the domain, SIPE

7

reduces the number of formulas in the system and makes deductions more efficient. There is currently no provision for creating objects as actions are executed. Some domains can be best represented as creating and destroying objects (e.g., after you have made an omellette do the original three eggs still exist as objects?), but SIPE does not support this type of representation.

3.2 Representation of operators

Operators representing actions the system may perform contain information about the objects that participate in the actions (represented as resources and arguments of the actions), what the action is attempting to achieve (its goal), the effects of the actions when they are performed, and the conditions necessary before the actions can be performed (their preconditions). Before SIPE's representation is described in detail some basic assumptions made by SIPE about the effects of actions need to be presented.

Determining the state of the world after actions have been performed (e.g., the planner must tell if goals have been achieved), involves solving the frame problem. Here we make what Waldinger [17] has called the STRIPS assumption which is that all relations mentioned in the world model are assumed to remain unchanged unless an action in the plan specifies that some relation has changed. In STRIPS, an action specifies that a relation has changed by mentioning it on an "addlist" or "deletelist". Alternatively, relations that change might be deduced from general frame axioms as long as the deduction is tightly controlled.

Making this assumption puts requirements on the formalism used for representing the domain since it must support the STRIPS assumption. While the STRIPS assumption may be very limiting in the representation of rich domains such as automatic programming, there are a many domains of interest for which it causes no problems. For example, the fairly simple environments in which robot arms often operate appear to be capable of being adequately represented in a system embodying the STRIPS assumption. SIPE currently makes the closed-world assumption: any negated predicate is true unless the unnegated form of the predicate

8

is explicitly given in the model or in the effects of an action that has been performed. This is not critical; the system could be changed to assume that a predicate's truth-value is unknown unless an explicit mention of the predicate is found in either negated or unnegated form. Deduction in SIPE does not violate the closed-world assumption; it is used only to deduce effects of an action when the action is added to a plan (thus saving the operator that represents the action from having to specify these effects).

Many features combine to make SIPE's operator description language an improvement over operator descriptions in previous systems. These features will be presented by discussing the example operator given in Figure 1, with the more important features having subsections devoted to them. The SIPE system has produced correct parallel plans for problems in four different domains, one of which is the blocks world (described in [10]) for which many domain independent planning systems (e.g., NONLIN and NOAH) have presented solutions. To facilitate comparison with these systems, a PUTON operator for the blocks world in the SIPE formalism is shown in Figure 1.

The operator's effects, preconditions, and purpose are all encoded as first-order predicates on variables and objects in the domain. (In this case, BLOCK1 and OBJECT1 are variables.) Negated predicates that occur in the effects of an operator essentially remove from the model a fact that was true before but is no longer true.

Operators contain a plot that specifies how the action is to be performed in terms of actions and goals at either the current level or some lower level of the hierarchy. Like plans, plots are represented as procedural networks. When used by the planning system, the plot can be viewed as instructions for expanding a node in the procedural network to a greater level of detail. The plot of an operator can be described either in terms of *goals* to be achieved (i.e., a predicate to make true), or in terms of *processes* to be invoked (i.e., an action to perform). (NOAH represented a process as a goal with only a single choice of action.) Encoding a step as a process implies that only the action it defines can be taken at that point, while encoding a step as a goal implies that any action can be taken that will achieve the goal. Another less

9

```
OPERATOR: PUTON
ARGUMENTS: BLOCK1, OBJECT1 IS NOT BLOCK1;
PURPOSE: (ON BLOCK1 OBJECT1);
PLOT:
    PARALLEL
        BRANCH 1:
            GOALS: (CLEARTOP OBJECT1);
            ARGUMENTS: OBJECT1;
        BRANCH 2:
            GOALS: (CLEARTOP BLOCK1);
            ARGUMENTS: BLOCK1;
    END PARALLEL

    PROCESS
    ACTION: PUTON.PRIMITIVE;
    ARGUMENTS: OBJECT1;
    RESOURCES: BLOCK1;
    EFFECTS: (ON BLOCK1 OBJECT1);
END
```

Figure 1
a PUTON operator in SIPE

explicit difference between encoding a step as a goal or as a process is whether the emphasis is on the situation to be achieved or the actual action being performed.

During planning, an operator is used to expand an already existing GOAL or PROCESS node in the procedural network to produce additional procedural network structure at the next level. For example, the PUTON operator might be applied to a GOAL node in a plan whose goal predicate is (ON A B). Operators may specify preconditions that must obtain in the world state before the operator can be applied. (The operator in Figure 1 has no precondition.) Operators contain lists of resources and arguments to be matched with the resources and arguments of the node being expanded. In our example, A and B in the GOAL node are matched with BLOCK1 and OBJECT1 in the PUTON operator when the operator is used to expand the node. The plot of the operator is used as a template for generating two GOAL nodes and one PROCESS node in the plan.

Operators in SIPE provide for posting of constraints on variables, specification of resources,

explicit representation of the purpose of each action, and the use of deduction to determine effects of actions. Each of these features is described below in some detail. In addition to these features, SIPE provides the ability to apply the plots of operators to lists of objects. Variables in an operator can be instantiated to a list of objects by calling a generator function given in the GENERATOR attribute of the CLASS node of the variable. For example, suppose BLOCKS1 is a variable in an operator and the CLASS node for BLOCKS in the type hierarchy has a function as the value of its GENERATOR attribute. In this case, application of that operator will result in the function being called to instantiate BLOCKS1 (e.g., to a list of blocks). The plot of the operator may then contain an ITERATE-BEGIN and an ITERATE-END, and the plot within these tokens will be reproduced in the resultant procedural net once for each different block in the list BLOCKS1. For example, a CLEARTOP goal could be generated for each block in the list. This is useful for generating paths to move along in the aircraft operations domain, as described in the section on performance.

3.2.1 Partially Described Objects

One of SIPE's most important advances over previous domain independent planning systems is its ability to construct partial descriptions of unspecified objects. This ability is important both for domain represention (e.g., objects with varying degrees of abstractness can be represented in the same formalism), and for efficiently finding solutions (since decisions can be delayed until partial descriptions provide more information). Comparisons to other systems are at the end of this section, but almost no previous domain independent planning systems have used this approach (e.g., NOAH cannot partially describe objects) so the constraints in SIPE will be documented in some detail.

Planning variables which do not *yet* have an instantiation (these are INDEFINITE nodes in the type hierarchy) can be partially described by setting constraints on the possible values an instantiation might take. Constraints may place restrictions on the properties on an object (e.g., requiring certain attribute values for it in the type hierarchy), and also require that

11

certain relationships exist between an object and other objects (e.g., predicates that must be satisfied in a certain world state). SIPE provides a general language for expressing these constraints on variable bindings so they can be encoded as part of the operator. During planning, the system also generates constraints based on interactions within a plan, propagates them to variables in related parts of the network, and finds variable bindings that satisfy all constraints.

The allowable constraints in SIPE on a variable V are listed below·

•CLASS. This constrains V to be in a specific class in the type hierarchy. In SIPE's operator description language there is implicit typing based on variable name, so in the PUTON operator in Figure 1 the variable created for BLOCK1 has a CLASS constraint which requires the instantiation for the variable to be a member of the class BLOCKS. Similarly, the OBJECT1 variable has a CLASS constraint for class OBJECTS.

•NOT-CLASS. V must be instantiated so that it is not a member of a given class.

•PRED. V must be instantiated so that a given predicate (in which V is an argument of the predicate), is true. This results in an explicit number of choices for V's instantiation since all true facts are known (by the closed-world assumption).

•NOT-PRED. V must be instantiated so that a given predicate (in which V is an argument of the predicate), is not true.

•INSTAN. V must be instantiated to a given object. This could be represented using SAME applied to objects as well as variables (or using PRED with an EQ predicate), but instantiation is a basic function of the system and warrants its own constraint to provide a slight gain in efficiency.

•NOT-INSTAN. V must not be instantiated to a given object.

•SAME. V must be instantiated to the same object to which some other given variable is instantiated.

12

•NOT-SAME. V must not be instantiated to the same object to which some other given variable is instantiated. In the PUTON operator in Figure 1, the phrase "IS NOT BLOCK1" results in a NOT-SAME constraint being posted on both BLOCK1 and OBJECT1 that requires they not be instantiated to the same thing. Thus if SIPE is looking for a place to put block A, it will not choose A as the place to put it.

•OPTIONAL-SAME. This is similar to SAME but merely specifies a preference and is not binding. For example, one would prefer to conserve resources by making two variables be the same object, but if this is not possible then different objects are acceptable.

•OPTIONAL-NOT-SAME. This is similar to NOT-SAME but not binding. If SIPE notices that a conflict will occur between two parallel actions if two variables are instantiated to the same object, then it will post a OPTIONAL-NOT-SAME constraint on both variables. If it is possible to instantiate them differently then a conflict is avoided. If it is not, they may be made the same but the system will have to correct the ensuing conflict (perhaps by not doing things in parallel).

•Any attribute name. This requires a specific value for a specific attribute of an object. For example, the PUTON operator could have specified "BLOCK1 WITH COLOR RED". This would create a constraint on BLOCK1 requiring the COLOR attribute (in the type hierarchy) of any possible instantiation to have the value RED. For attributes with numerical values, "greater than" and "less than" can also be used. In planning an airline schedule, for example, the operator used for cross country flights might contain the following variable declaration: "PLANE1 WITH RANGE GREATER THAN 3000".

This list documents the constraints used in SIPE. Constraints add considerably to the complexity of the planner since they interact with all parts of the system. For example, to determine if a goal predicate is true, SIPE must see if it matches predicates that are effects earlier in the plan. This may involve matching two variables which are arguments to the two predicates, and this in turn involves determining whether the constraints on the two variables

13

are compatible. In a similar way, constraints also interact with the deductive capability of the system (to be described later). Constraints also affect critics since determining if two concurrent actions interact may depend on whether their constraints are compatible. SIPE must also solve a general constraint satisfaction problem with reasonable efficiency, though how to control the amount of processing spent on constraint satisfaction is an open and important question.

Use of constraints is a major advance over previous domain independent planning systems. NOAH, for example, would have to represent every property of an object as a predicate and then have each such predicate as either a precondition of an operator or a goal in the plan in order to get variables properly instantiated. In SIPE an operator might declare a variable as "CARGOPLANE1 WITH RANGE 3000" and the plan using this variable can assume it has the proper type of aircraft. In NOAH, goals similar to (CARGOPLANE X) and (RANGE X 3000) would have to be included in the operator and achieved as part of the plan. This makes both the operators and plans much longer and harder to use and understand. In addition to syntactic sugar, constraints in SIPE improve efficiency and expressibility. The OPTIONAL-SAME and OPTIONAL-NOT-SAME constraints used in resource reasoning cannot be expressed as goals or preconditions in a system like NOAH. The constraint satisfaction algorithm used in SIPE takes advantage of the fact that invariant properties of objects are stored directly in the type hierarchy. The lookup of such properties in SIPE is much more efficient than the process of looking through the plan to determine which predicates are currently true, as would have to be done in systems like NOAH and NONLIN.

Some domain dependent systems make use of constraints. Stefik's system [12], one of the few existing planning systems with the ability to construct partial descriptions of an object without identifying the object, operates in the domain of molecular genetics. Our system extends Stefik's approach in three ways. (1) We provide an explicit, general set of constraint that can be used in many domains. Stefik does not present a list of allowable constraints in his system, and some of them that are mentioned seem specific to the genetics domain.

14

(2) Constraints on variables can be evaluated before the variables are fully instantiated. For example, a set can be created that can be constrained to be only bolts, then to be longer than one inch and shorter than two inches, and then to have hex heads. This set can be used in planning before its members are identified in the domain. (3) Partial descriptions can vary with the context, thus permitting simultaneous consideration of alternative plans involving the same unidentified objects. This is described in more detail in the section on search control.

3.2.2 Resources

The formalism for representing operators in SIPE includes a means of specifying that some of the variables associated with an action or goal actually serve as resources for that action or goal (e.g., BLOCK1 is declared as a resource in the PUTON.PRIMITIVE action of the PUTON operator in Figure 1). Resources are to be employed during a particular action and then released, just as a frying pan is used while sauteeing vegetables. Reasoning about resources is a common phenomenon. It is a useful way of representing many domains, a natural way for humans to think about problems, and, consequently, an important aid to interaction with the system.

SIPE has specialized knowledge for handling resources; declaration of a resource associated with an action is a way of saying that one precondition of the action is that the resource be available. Mechanisms in the planning system, as they allocate and deallocate resources, automatically check for resource conflicts and ensure that these availability preconditions will be satisfied. One advantage of resources, therefore, is that they help in the axiomatization and representation of domains. The user of the planning system does not have to axiomatize as a precondition the availability of resources in the domain operators. (Such an axiomatization may be difficult since the critics must correctly use the representation to recognize problems with unavailable resources.) This enables both SIPE's operators and plans to be shorter and easier to understand than similar operators and plans in domain independent parallel planning systems such as NOAH and NONLIN. Resource availability in the latter would have to be

15

axiomatized and checked in the preconditions of operators, while resource conflicts (which can be quickly identified and corrected in SIPE) would have to be caught by the normal problematic interaction detector, which is less efficient (as shown in the section on parallel interactions).

3.2.3 Purposes

In the procedural networks that represent plans, PROCESS and GOAL nodes represent an action to be performed or a goal to be achieved. Associated with these nodes are predicates stating the expected effects of performing the action or achieving the goal. When a node is planned to a greater level of detail by applying an operator, the expansion may consist of many nodes. Which node in the expansion achieves the main purpose of that sequence of steps must be determined to ascertain when the effects of the higher level node become true in the more detailed expansion. (This may be the last in a series of nodes that, acting together, achieve the tacit "purpose" of the expansion.) Determining purposes correctly is also necessary for correcting problems arising from parallel interactions.

The word *purpose* will be used to refer to different objects in the following discussion; the meaning is disambiguated by context. The purpose of an operator is its purpose attribute, which is a conjunction of predicates. When the operator is used to produce an expansion, this purpose attribute is used to determine the purpose of the expansion that is the node in the piece of procedural net produced by the expansion that achieves the purpose of the operator. Nodes in a procedural net may also have a purpose, which is another node (occurring later) in the procedural net.

Let us consider the GOAL node in Figure 2 in which the goal is (SECURED PUMP). The SECURE PUMPBOLTS operator expands this node into three nodes at the next level of detail as shown. The first node might be called a preparatory action, and the last a cleanup action. Somewhere must be encoded the fact that (SECURED PUMP) becomes true after the second node in the expansion. This is needed, for example, in answering user questions or determining

16

goal node in
procedural net:

```
GOAL
arguments: PUMP
goal: (SECURED PUMP)
```

Operator node from type hierarchy (not part of the procedural net):

```
OPERATOR
SECURE PUMPBOLT
goal: (SECURED PUMP)
arguments: PUMP1
resources: WRENCH1
purpose: (TIGHT PUMPBOLTS)
plot: ...
```

Expansion produced after applying SECURE PUMPBOLT operator to goal node:

```
PROCESS
GET WRENCH
effects:
(IN WRENCH HAND)
(NOT (IN WRENCH TOOLBOX))
```

```
PROCESS
TIGHTEN BOLTS
effects:
(TIGHT PUMPBOLTS)
```

```
PROCESS
PUT WRENCH AWAY
effects:
(IN WRENCH TOOLBOX)
(NOT (IN WRENCH HAND))
```

Figure 2
Using an Operator to Produce an Expansion

the correct world state at the PUT WRENCH AWAY node (perhaps some operators for putting

the wrench away may be affected by or depend upon the state of the pump). Each operator in

SIPE has a PURPOSE attribute that specifies a conjunction of predicates, which is the main

purpose of any expansion produced with this operator. (The PURPOSE attribute is also used

to determine when to apply the operator whenever no GOAL attribute is given). Thus, the

PURPOSE of the SECURE PUMPBOLTS operator is the predicate (TIGHT PUMPBOLTS).

This is included in the effects of the TIGHTEN BOLTS node produced in the plan, so the

(SECURED PUMP) effect is copied down to this node. In NOAH the assumption was that the

last node of an expansion achieved the main purpose, so the effects were copied down to that

node. In the example above, this would incorrectly attach (SECURED PUMP) to the PUT

WRENCH AWAY node. SIPE allows flexibility in specifying purposes, so that situations like

17

the one described above can be represented accurately.

Besides determining purposes between hierarchical levels (as above), it is helpful to determine purposes within a level. For example, in Figure 2 it would be helpful to know that the purpose of the GET WRENCH node in the plan is the TIGHTEN BOLTS node. If there were a conflict over the wrench with some parallel action, the system would then be able to determine that the wrench could be released to another process after the pumpbolts were tight and before the wrench was put away. SIPE also provides flexibility in specifying these purposes. Nodes in the plot of an operator can specifically mention later nodes in the plot as their purpose. If none is mentioned, then the default (which takes advantage of the flexibility in specifying purposes between hierarchical levels) is that the purpose of all nodes in the expansion before the main purpose of the expansion (TIGHTEN BOLTS in this case) is that main purpose. This default correctly fixes the purpose of the GET WRENCH node in Figure 2, while in NOAH the default purpose would imply the wrench must be kept until it is back in the tool box.

Correctly determining purposes is important for correcting problematic parallel interactions within a plan. The section on parallel interactions discusses in some detail the role purposes play in correcting problems.

3.3 Deductive Operators

In addition to operators describing actions, SIPE allows specification of deductive operators that deduce facts from the current world state. As more complex domains are represented, it becomes increasingly important to deduce effects of actions from axioms about the world rather than explicitly representing these effects in operators. For example, the PUTON operator in Figure 1 lists only (ON BLOCK1 OBJECT1) as an effect. It does not mention which objects are and are not now CLEARTOP since that is deduced by deductive operators. Deductive operators in SIPE may include both existential and universal quantifiers, and so provide a rich formalism for deducing (possibly conditional) effects of an action. Effects that are deduced in SIPE are considered to be side effects. (Operators can also specify effects as either main

18

```
DEDUCTIVE.OPERATOR: DCLEAR
ARGUMENTS: OBJECT1,OBJECT2,BLOCK3 IS NOT OBJECT2,
    OBJECT4 CLASS EXISTENTIAL IS NOT OBJECT1;
TRIGGER: (ON OBJECT1 OBJECT2);
PRECONDITION: (ON OBJECT1 BLOCK3), (NOT (ON OBJECT4 BLOCK3));
EFFECTS: (CLEAR BLOCK3);
```

Figure 3
a deductive operator in SIPE

effects or side effects.) Knowing which effects are merely side effects is important in handling parallel actions (see next section).

Figure 3 shows one of the deductive operators in the SIPE blocks world for deducing CLEARTOP relationships. Deductive operators are written in the same formalism as other operators in SIPE, permitting the system to control deduction with the same mechanisms it uses to control the application of operators. This also allows constraints to be used and, as this example shows, they play a major role in SIPE's deductive capability.

Deductive operators have triggers for controlling their application. The DCLEAR operator in Figure 3 is applied when OBJECT1 is placed on OBJECT2. Deductive operators have no instructions for expanding a node to a greater level of detail. Instead, if the precondition of a deductive operator holds, its effects can be added to the world model (in the same context in which the precondition matched) without changing the existing plan. This may "achieve" some goal in the plan (by deducing that it has already been achieved), and avoid the need to plan actions to achieve it. In Figure 3, matching the precondition will bind BLOCK3 to the block that OBJECT1 was on before it moved to OBJECT2. Since OBJECT4 is constrained to be in the EXISTENTIAL class (SIPE's way of specifying existentially quantified variables) and is constrained to not be OBJECT1, the precondition will match (and CLEARTOP of BLOCK3 deduced) only if OBJECT1 is the only object on BLOCK3 (at the time just before moving OBJECT1 to OBJECT2).

Besides simplifying operators, deductive operators are important in many domains for their ability to represent conditional effects. In NOAH's blocks world, only one block may be

on top of another so that whenever a block is moved, the operator for the move action can be written to explicitly state the effect that the block underneath will be clear. In the more general case in which one large block might have many smaller blocks on top of it, there may or may not be another block on the underneath block so the effects of the action must be conditional on this. Since systems like NOAH and NONLIN must mention effects explicitly (universally or existentially quantified variables are not allowed in the description of effects), they cannot represent this more general case with a single move operator. These systems would need two move operators - one for the one-block-on-top case, and another for the many-blocks-on-top case, and the preconditions to separate the cases would add undesirable complication to the representation of the operators.

As the above example shows, SIPE's deductive operators allow existential quantifiers and are powerful enough to handle this case. In the blocks world, SIPE can deduce all the clearing and unclearing effects that occur, so the operators themselves do not need to represent these effects. As domains grow to include many operators, this becomes very convenient. Deductive operators provide a way to distinguish side effects, which can be important. Using deduction, more complicated blocks worlds can be represented more elegantly in SIPE than in previous domain independent planners.

4. Parallel Interactions

As noted before, parallelism is considered beneficial since optimal plans in many domains require it. (Two segments of a plan are in parallel if the partial ordering of the plan does not specify that one segment must be done before the other.) The approach used in SIPE, therefore, is to keep as much parallelism as possible and then detect and respond to interactions between parallel branches of a plan. There are three aspects to this situation: recognizing interactions between branches; correcting harmful interactions that keep the plan from accomplishing its overall goal; taking advantage of helpful interactions on parallel branches so as not to produce

20

inefficient plans.

This section first defines helpful and harmful interactions, and then describes new features and heuristics in SIPE that aid in handling them. These fall into four areas: (1) reasoning about resources, which is the major contribution of SIPE; (2) using constraints to generate correct parallel plans; (3) explicitly representing the purpose of each action and goal to help solve harmful interactions correctly; (4) taking advantage of helpful interactions.

SIPE's abilities to handle parallel actions are best described in the context of a sample problem. The canonical simple problem for thinking about parallel interactions is the three-blocks problem. Blocks A, B, and C are on a table or on one another. The goal is to achieve (ON A B) in conjunction with (ON B C), thus making a three-block tower. (Initially the two goals are represented as being in parallel.) To move the blocks there is a PUTON operator (see Figure 1, for example) that puts OBJECT1 on OBJECT2. It specifies the goals of making both OBJECT1 and OBJECT2 clear before performing a primitive move action. (The table is assumed always to be clear and a block is clear only when no block is on top of it.) This problem will be used below to provide examples of interactions.

4.1 Defining Helpful and Harmful Interactions

If two branches of a plan are in parallel, an interaction is defined to occur when a goal that is trying to be achieved in one branch (at any level in the hierarchy) is made either true or false by an action in the other branch. Since the actions in a plan explicitly list their effects (a feature shared by many planners such as NOAH and NONLIN), it is always possible to recognize such interactions. (In a hierarchical planner, however, they may not appear until lower levels of the hierarchy of both branches have been planned.) By requiring that a goal be involved in the interaction, we attempt to eliminate interactions that we do not care about. For this to succeed, the domain must be encoded so that all important relationships are represented as goals at some level. As we shall see later, this is reasonable in SIPE.

The planner can possibly take advantage of a situation in which a goal in one branch

21

is made true in another branch (a helpful interaction). Suppose we solve the three-blocks problem, starting with A and C on the table and B on A. In solving the (ON B C) parallel branch, the planner will plan to move B onto C, thus making A clear and C not clear. Now, while an attempt is made to move A onto B in the (ON A B) branch, the goal of making A clear becomes part of the plan. Since A is not clear in the initial state, the planner may decide to make it true by moving B from A to the table (after which it will move A onto B). In this case it would be better to recognize the helpful effect of making A clear, which happens in the parallel branch. Then the planner could decide to do (ON B C) first, after which both A and B are clear and the (ON A B) goal is easily accomplished.

The planner must decide whether or not to add more ordering constraints to the plan to take advantage of such helpful interactions. Ordering the parallel branches sequentially is the best solution to this problem because (ON B C) must be done first in any case, but in other problems an ordering suggested to take advantage of helpful effects may be the wrong thing to do from the standpoint of eventually achieving the overall goal. In general, the planner cannot make such an ordering decision without error unless it completely investigates all the consequences of such a decision. Since this is not always practical or desirable, planning systems use heuristics to make such decisions.

If an interaction is detected that makes a goal false in a parallel branch, there is a problematic (i.e., possibly harmful) interaction which may mean that the plan is not a valid solution. For example, suppose the planner does not recognize the helpful interaction in our problem and proceeds to plan to put B on the table and A on B in the (ON A B) branch. The plan is no longer a valid solution (if it is assumed that one of the two parallel branches will be executed before the other). The planner must recognize this by detecting the problematic interaction. Namely, the goal of having B clear in the (ON B C) branch is made false in the (ON A B) branch when A is put onto B. The planner must then decide how to rectify this situation.

As with helpful interactions, there is no easy way to solve harmful interactions. Here too

22

a correct solution may require that all future consequences of an ordering decision be explored. Stratagems other than ordering may be necessary to solve the problem. For example, a new operator may perhaps need to be applied at a higher level. Consider the problem of switching the values of the two registers in a two-register machine. Applying the register-to-register move operator creates a harmful interaction that no ordering can solve, since a value is destroyed. The solution to this interaction involves applying a register-to-memory move operator at a high level in order to store one of the values temporarily. Correcting many types of harmful interactions efficiently seems very difficult in a domain independent planner – domain specific heuristics may be required.

4.2 Reasoning about Resources

The formalism for representing operators in SIPE includes a means of specifying that some of the variables associated with an action or goal actually serve as resources for that action or goal. As we have seen, one advantage of resources is that they help in the axiomatization and representation of domains. Another important advantage of resources is that they help in early detection of problematic interactions on parallel branches. The system does not allow one branch to use an object which is a resource in a parallel branch.

The above example of achieving (ON A B) and (ON B C) as a conjunction shows how SIPE uses resource reasoning to help with parallel interactions. Figure 4 depicts a plan that might be produced by NOAH or NONLIN (or SIPE without making use of resource reasoning) for this problem. Figure 5 shows a plan from SIPE using resources in the operators.

In NOAH and NONLIN, both original GOAL nodes are expanded with the PUTON operator or its equivalent. This produces a plan similar to the one shown in Figure 4. The central problem is to be aware that B must be put on C before A is put on B (otherwise B will not be clear when it is to be moved onto C). NOAH and NONLIN both build up a table of multiple effects (TOME) that tabulates every predicate instance listed as an effect in the parallel expansions of the two GOAL nodes. Using this table, the programs detect that B is

23

Figure 4
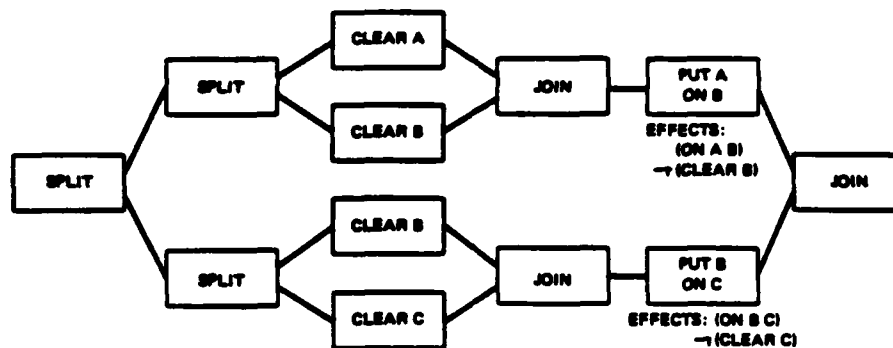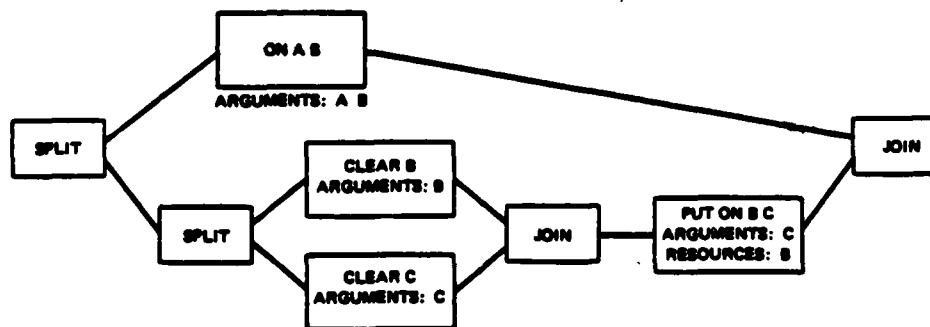a plan without resources

Figure 5
a plan with resources

made clear in the expansion of (ON B C), but is made not clear in the (ON A B) expansion.
Both programs then solve this problem by doing (ON B C) first.

SIPE uses its resource heuristic to detect this problem and propose the solution without
having to generate a TOME. (SIPE does do a TOME-like analysis to detect interactions that

24

do not fit into the resource reasoning paradigm.) When some object is listed in an action as a resource, the system then prevents that particular object from being mentioned as either a resource or an argument in any action or goal that is in parallel. In the PUTON operator, the block being moved is listed as a resource in the primitive PUTON action because it is being physically moved. (Therefore, nothing in a parallel branch should try to move it–or even be dependent on its current location.) Thus, as soon as the expansion of (ON B C) with the PUTON operator is accomplished and the plan in Figure 5 produced, SIPE recognizes that the plan is invalid since B is a resource in the expansion of (ON B C) and an argument in (ON A B). This can de detected without expanding the (ON A B) goal at all and without generating a TOME.

Not allowing a resource to be mentioned as either a resource or an argument in any action or goal that is in parallel is a strong restriction (though useful in practice). This is sometimes too strong a restriction, so SIPE also permits the specification of shared resources, whereby a resource in one branch can be an argument in a parallel branch, but not a resource. Ideally we would like to have predicates specifying sharing conditions, but this has not yet been implemented.

Resources help in solving harmful interactions, as well as in detecting them. In resource conflicts, no goal is made false on a parallel branch; however, if the resource availability requirements were axiomatized as precondition or goal predicates, an availability goal would be made false on a parallel branch. Thus, resource conflicts are considered to be harmful interactions. SIPE uses a heuristic for solving resource-argument conflicts. Such an interaction occurs when a resource in one parallel branch is used as an argume in another parallel branch (distinct from a resource-resource conflict, in which the same object is used as a resource in two parallel branches). This is the type of conflict that occurs in the plan in Figure 5, since B is a resource in the primitive PUTON action and an argument in (ON A B).

SIPE's heuristic for solving a resource-argument conflict is to put the branch using the object as a resource before the parallel branch using the same object as an argument. In this

way SIPE decides that (ON B C) must come before (ON A B) in Figure 5. This is done without generating a TOME, without expanding the original (ON A B) node, and without analyzing the interaction. The assumption is that an object used as a resource will have its state or location changed by such use; consequently, the associated action must be done first to ensure that it will be "in place" when later actions occur that use it as an argument. This heuristic is not guaranteed to be correct, but it has proven useful in the four domains encoded in SIPE. (The user can easily prevent the employment of this heuristic by interacting with the system.)

Many interactions that would be harmful in the other systems are dealt with in SIPE by the resource-reasoning mechanisms. To take full advantage of resources, the system posts constraints. This capability is discussed briefly below.

4.3 Constraints and Resource Assignment

SIPE can accumulate various constraints on unbound variables in a plan, which is useful for taking full advantage of resources to avoid harmful interactions. When variables that are not fully instantiated are listed as resources, the system posts constraints on the variables that point to other variables that are potential resource conflicts. When allocating resources, the system then attempts to instantiate variables so that no resource conflicts will occur. (See the OPTIONAL-NOT-SAME constraint in section 3.) For example, if a robot arm is used as a resource in the block-moving operators, the system will try to use different robot arms (if they are available) on parallel branches, thus avoiding resource conflicts. If only one arm is available, it will be assigned to both parallel branches in the hope that the plan can later be ordered to resolve the conflict. In this way many harmful interactions are averted by intelligent assignment of resources.

4.4 Solving Harmful Interactions

The difficulty entailed in eliminating harmful interactions has already been discussed. However, if the system knows why each part of the plan is present, it can use this information

to come up with reasonable solutions to some harmful interactions. Suppose a particular predicate is made false at some node on one parallel branch and true at another node on another parallel branch. Depending on the rationale for including these nodes in the plan, it may be the case that the predicate is not relevant to the plan (an extraneous side effect), or must be kept permanently true (the purpose of the plan), or must be kept only temporarily true (a precondition for later achievement of a purpose). SIPE's ability to specify purposes flexibly and to separate side effects from main effects enables it to distinguish accurately these 3 cases, and, therefore, to find more reasonable solutions than could most previous planners.

Solutions to a harmful interaction may depend on which of these cases holds. Let us call the three cases side effect, purpose, and precondition, respectively, and analyze the consequent possibilities. If the predicate in conflict on one branch is a precondition, one possible solution is to further order the plan, first doing the part of the plan from the precondition on through its corresponding purpose. Once this purpose has been accomplished, there will be no problem in negating the precondition later. This solution applies no matter which of the three cases applies to the predicate in the other conflicting branch.

In case both conflicting predicates are side effects, it is immaterial to us if the truth value of the predicate changes and thus no real conflict exists. In the case of a side effect that conflicts with a purpose, one solution is to order the plan so that the side effect occurs before the purpose; thus, on_e the purpose has been accomplished it will remain true. When both conflicting predicates are purposes, there is no possible ordering that will achieve both purposes at the end of the plan. The planner must use a different operator at a higher level or plan to reachieve one of the purposes later. However, none of the above suggestions for dealing with interactions can be guaranteed to produce the best (by some metric, e.g., shortest) solution.

This has been a brief summary of SIPE's algorithm for dealing with harmful interactions. Systems like NOAH and NONLIN do similar things. However, SIPE provides methods for more precise and efficient detection. It should be emphasized that many interactions that

27

would be problematical in the other systems are dealt with in SIPE by the resource-reasoning mechanisms and therefore do not need to be analyzed. When interactions are being analyzed, SIPE requires that one of the conflicting predicates be a goal (not just a side effect) at some level in the hierarchy. In this way, interactions between side effects that pose no problems are not even detected. This requires that all important predicates be recognized as goals at some level, which is easily done in SIPE's hierarchical planning scheme. SIPE provides for exact expression of the purpose of any goal in its operators which again leads to better analysis of interactions. The system also distinguishes between main and side effects at each node in the plan. This makes it easy to tell which predicates are of interest to us at any level of the plan without looking up the hierarchy (since higher-level goals will become main effects at lower-level actions).

4.5 Achieving Goals Through Linearization

SIPE recognizes helpful interactions and will try to further order the plan to take advantage of them, although the user can control this interactively if he wishes. If a goal that must be made true on one parallel branch is actually made true on another parallel branch, the system will under certain conditions order the plan so that the other branch occurs first (if this causes no other conflicts).

NOAH was not able to take advantage of such helpful effects. NONLIN did have an ability to order the plan in this way. This is an important ability in many real-world domains, since helpful side effects occur frequently. For example, if parallel actions in a robot world both require the same tool, only one branch need plan to get the tool out of the tool box; the other branch should be able to recognize that the tool is already out on the table.

5. Search control

The automatic search in SIPE is a simple depth-first search to a given depth limit

28

without using knowledge for making choices or backtracking. Obviously, it does not perform particularly well on large problems. The problems involved in planning at the meta-level to control the search are discussed below. The poor performance of automatic search is not debilitating in SIPE, since it has been designed and built to support interactive planning. The user can easily invoke planning operations at any level without being required to make tedious choices that could be performed automatically. In SIPE the user can direct low-level and specific planning operations (e.g., "instantiate PLANE1 to N2636G", "expand NODE 32 with the PUTON operator"), high-level operations that combine these lower-level ones (e.g., "expand the whole plan one more level and correct any problems"), or operations at any level between the two (e.g., "assign resources", "expand NODE 32 with any operator", "find and correct harmful interactions").

The examples above need not be given to SIPE as text since the interactive component of SIPE makes use of graphics and has been implemented on a high-resolution black-and-white bitmap display and a color-graphics terminal [8].[2] The planning choices available to the user appear in a menu from which one can be selected by pointing with a mouse or joystick. Similarly, steps in a plan (nodes in the network) can either be referred to by name or by pointing to them.

Both the procedural networks (plans) produced and, for certain domains, the domain configuration (principally the location of objects) are presented graphically. The user can choose to view different portions of the plan, at different levels of detail, and can look at any alternative plans. The user can also see a graphic representation of either the actual domain configuration or the one that will exist after some sequence of planned steps. The configuration is generally shown together with a plan or partial plan, and the configuration corresponds to the expected state following execution of that plan.

One feature of SIPE not found in previous domain independent planners is the ability to

explore alternatives in parallel. This is advantageous in an interactive environment since it allows the user to conduct a best-first search easily. In addition to supporting breadth-first and depth-first planning, the interactive planning operations allow *islands* to be constructed in a plan (to arbitrary levels of detail), and then linked together later. The following sections describe the implementation of parallel alternatives, and the use of meta-planning to control the automatic search.

5.1 Exploring Alternatives in Parallel

A context mechanism has been developed to allow constraints on a variable's value to be established relative to specific plan steps. Constraints on a variable's value, as well as its instantiation (possibly determined during the solution of a general constraint-satisfaction problem), can be retrieved only relative to a particular context. This permits the user to shift focus back and forth easily between alternatives.

SIPE accomplishes this in a hierarchical procedural-network paradigm by introducing CHOICE nodes in the procedural networks at each place an alternative can occur. Constraints are stored relative to choice points. Thus, the constraints on a variable at a given point in a plan can be accessed by specifying the path of choices in the plan that is to be followed to reach that point. Different constraints can be retrieved by specifying a different plan (path of choices). This shifting of focus between alternatives cannot be done in systems using a backtracking algorithm, in which descriptions built up during expansion of one alternative are removed during the backtracking process before another alternative is investigated. Most other planning systems either do not allow alternatives (e.g., NOAH), or use a backtracking algorithm (e.g., Stefik's MOLGEN, NONLIN). An exception is the system described by Hayes-Roth et al. [5], in which a blackboard model is used to allow shifting focus between alternatives.

5.2 Meta-planning

Planning systems are continually making choices in an attempt to find a plan, and they

30

have some *control structure* that controls the making of these decisions. The term *meta-planning* is widely used for referring to reasoning about the planning process itself. Thus the control structure of a planning program is doing meta-planning, though in SIPE only in an uninteresting way since the control structure is simple and contains little knowledge. (Of course, in SIPE the most abstract operators could be written to make use of meta-planning knowledge in a particular domain.) Many researchers have realized that meta-planning is an important element in being able to control the search efficiently, pointing out that planning about the planning process can be done in the same way as planning about the domain, enabling one system architecture to be used for both the planning system and its control structure. As Pat Hayes says in [4]: "We need to be able to describe processing strategies in a language at least as rich as that in which we describe the external domains, and for good engineering, it should be the same language."

Despite all the talk about meta-planning, no domain independent planner has done interesting meta-planning. This section shows why this is so, but first the idea of meta-planning must be clarified because the term is used in a vague way by many people. Once meta-planning has been described, the problems posed by meta planning for domain independent planners will be presented, and finally the accomplishments of previous planning systems with respect to these new ideas are described.

The above definition is vague, and the meta-planning process needs to be described in more detail. It will be shown that one problem in being more precise about meta-planning is that there is often no clear dividing line between the external domain and the planning process (counter to what Wilensky argues in [19]). Given any particular system, it will likely be obvious what is at a meta-level in that system. But any particular piece of knowledge might be encoded at either the meta-level or the domain level and it is not always clear which is best. It is trivial to make any domain operator into a meta-operator, and many meta-operators can probably be wired into the domain during design of the domain representation. A later example demonstrates an operator that would be reasonable at either the domain or

31

meta-levels.

Wilensky claims that meta-planning can be distinguished by the fact that "meta-goals are declarative structures", but certainly all goals in a planner can be declarative. He claims that another distinguishing characteristic is "meta-goals are domain independent, encoding only knowledge about planning in general". It is argued below that meta-goals actually contain varying amounts of domain dependent knowledge. Furthermore this is necessary since good planning strategies may differ for differing domains.

Higher level domain knowledge generally blends into search control knowledge, and there is no clear dividing line between the two. One can view the whole system as a series of concentric circles with the most primitive external domain level in the center. As you move out to the next layer (concentric circle), the knowledge is at a higher level, more abstract, and possibly more (but perhaps less?) domain independent. There is no clear dividing line between the domain independence and domain dependence. It might be possible for a domain specific layer to encompass a more domain independent layer. It also appears there will be no clear dividing line between what should be meta-level knowledge and what should be domain level knowledge. (Again, in any particular system, it may be obvious what has been implemented at the meta-level.)

Some examples from house building should help support these claims. At the innermost layer, knowledge about nails and lumber is certainly domain specific. A few layers out, knowledge that the foundation should come before the roof is still fairly domain specific. Perhaps the next layer out contains advice such as "use existing objects". This is a fairly domain independent idea that is used by Sacerdoti in NOAH [10], and mentioned by Wilensky as a meta-goal for meta-planning. However, this idea still involves domain knowledge. It is good in the building domain to use the same piece of lumber to support the roof and to support the sheetrock on the walls. But in another domain, this may not be a good strategy. On the space shuttle, one may want different functions to be performed by different objects so the plan will be more robust and less endangered by the failure of any one object. So the "use

32

existing objects" idea makes assumptions about the domain that need to be stated. (Perhaps one only wants to use this idea for certain parts of the domain.) It would be reasonable to design a system where such an idea was implemented at the lowest planning level and talked about domain objects. It would also be reasonable to use this idea in the meta-planner as advice to the "instantiate-variable" planning operator.

At a still more abstract layer, the system may have advice such as "apply the operator that has been successful in the past". This is clearly control knowledge and seems not to be domain specific. However, the operator referred to is unspecified and will only become instantiated when this meta-operator is applied in a particular domain that contains information on past performance of operators. Furthermore, the effectiveness of this operator will depend on the characteristics of the domain as they are reflected in the performance data.

To summarize, meta-knowledge contains varying amounts of domain knowledge, or at least makes assumptions about the domain that should be made explicit. There is no sharp, clear difference between meta-planning and planning in the external domain. This means that what we are interested in is how domain specific a particular meta-planning technique is and how it can be used in other domains. There are clearly some useful meta-operators that tend towards being domain independent, for example: "evaluate (with domain knowledge?) which operator is most likely to succeed", "do not waste resources", "switch to more constraint checking when constraints are failing often", "switch to less constraint checking when much time is being spent checking constraints and very few failures or successes occur".

The remainder of this section describes what current planning systems have accomplished in the area of meta-planning. SIPE and other domain independent planners do not do interesting meta-planning, and the first subsection describes why this is so. Logical formalisms overcome some of the difficulties and the second subsection briefly mentions them. The third subsection describes some contributions of domain dependent systems to meta-planning.

33

5.2.1 Domain independent planners

There are several domain independent planners in the literature, but none of them does interesting meta-planning. Systems such as SIPE that do hierarchical planning can use abstract operators to encode some meta-planning knowledge, but, as the next paragraph argues, the most interesting meta-planning ideas cannot be encoding in this manner. The only other meta-planning done in such systems is obscure search control code; no meta-knowledge is expressed in the domain independent formalisms used in these systems for planning in the external domain. There is good reason for this, as these domain independent formalisms are not adequate for expressing interesting meta-knowledge.

These formalisms generally use a *model approach* for representing the domain. The essential characteristic of this approach is that all relationships that hold in the domain are expressed directly (e.g., disjunctions are not allowed in general), in the sense that the model can be queried in a lookup manner to quickly return an answer about the truth value of a relationship. This efficient querying ability is, of course, the motivation for the model approach. This approach also means that add and delete lists can be used to solve the frame problem. The disadvantage of the model approach is that many things cannot be represented since they do not admit to such direct representation. In particular, what we need to say about plans at a meta-level cannot easily fit into the model approach, since it will not be reasonable to represent explicitly (for example) every property of a plan, a failed search branch, an operator, or a constraint that we might want to reason about at the meta-level. The point is that, in these systems, the domain language is not all that rich (since it must satisfy this model approach), so in fact a reasonable language for meta-planning must be richer than commonly used domain languages.

5.2.2 Logic

The alternative to the model approach is to use formulas, say in some extension of standard first-order logic. Whatever can be expressed in the logic can be said in these formulas.

The disadvantage of the logic approach is that it may require an arbitrarily hard deduction to query the model. In addition, the frame problem is harder to solve. Actually most of the domain independent planners use a hybrid approach – they throw in as much logic as they can without losing efficiency in querying the model. Logic does, however, provide a rich enough language for doing interesting meta-planning, as is evidenced by that fact that Weyhrauch's FOL system [18] has the power to do sophisticated meta-reasoning. Many approaches have been outlined for doing planning in logic (or something similar), e.g., de Kleer [2], McDermott [6], and Rosenschein [9], however, none of these has been successful (i.e., has produced a program which performs adequately with reasonable resources) in a complex domain. The problem is insufficient control of the many deductions that the system can perform. De Kleer et al. have expressed some meta-knowledge in their formalism with apparent success on toy problems; however, the problem of expressing meta-knowledge and using it effectively in these systems remains largely unsolved.

5.2.3 Domain dependent planners

There are many domain dependent planners, e.g., PARADISE produces expert plans in chess middle games [20]. Of these, few do interesting meta-planning, Stefik's MOLGEN program [13] being an exception. Looking at Stefik's program then, the question is how many of the meta-planning ideas can be used in other domains. MOLGEN has two planning layers more abstract than the primitive domain laboratory steps, called the strategy space and design space. The idea behind the design space is that planning can be done as operations on constraints. This leads to meta-operators such as "propagate constraints" and "satisfy constraints". This is a useful and powerful domain independent idea that can easily be transferred to other domains.

However, nearly everything else in the design space is fairly domain specific. For example, in [5] we read that "FIND-UNUSUAL-FEATURES is a design operator that examines laboratory goals. Sometimes a good way to select abstract operators to synthesize objects in

35

cloning experiments is to find features in which the objects are highly specialized or atypical, and then find operators that act on those features." This idea is fairly specific to the genetics domain, and its implementation is probably completely specific. It is not clear that Stefik provides any formalism for expressing meta-knowledge or planning knowledge, leaving the impression that the meta-operators are arbitrary code: "PREDICT-RESULTS is the design operator for simulating the results of a proposed laboratory step. It activates a simulation model associated with each laboratory operator."

In the strategy space, there are operators to change focus, make choices, and undo these choices after they fail. These are certainly domain independent ideas, but not terribly interesting as making choices and undoing them is the idea behind simple backtracking (though they are elegantly incorporated in the system). It would be nice to have some meta-knowledge that analyzed the reason for failure and made the correct choice based on this analysis. Stefik provides a layered agenda mechanism for scheduling tasks from different planning layers that is certainly useful in many domains.

So the interesting contributions to meta-planning for other domains are the agenda mechanism and the idea of using constraints in this way. These are significant contributions, especially since they are part of a system that actually works on a hard problem, but much is left undone and most of the interesting meta-planning ideas have yet to be used successfully. Representative of these ideas are the following: analyzing failed search branches to guide the system to the correct backtrack point and the correct choice, analyzing operators using knowledge to choose the correct one, integrating new knowledge that is discovered during the planning or execution process, and changing system behavior as characteristics of the problem space become apparent (e.g., increase constraint checking if constraint tightness is high, or change the backtracking depending on the solution density).

6. Execution Monitoring

In real-world domains, things do not always proceed as planned. Therefore, it is desirable to develop better execution monitoring techniques and better capabilities to replan when things do not go as expected. This may involve planning for tests to determine if things are going as expected. Such tests may be expensive (e.g., taking a picture with a computer vision system) so care must be taken in deciding when to use them. The problem of replanning is also critical. In complex domains it becomes increasingly important to use as much as possible of the old plan, rather than to start all over when things go wrong.

SIPE has addressed only some of the problems of execution monitoring, and research is continuing in this area. During execution of a plan in SIPE, some person or computer system monitoring the execution can specify what actions have been performed and what changes have occurred in the domain being modeled. Based on this, the plan can be updated interactively to cope with unanticipated occurrences. Planning and plan execution can be intermixed by producing a plan for part of an activity and then executing some or all of that plan before elaborating on the remaining portion.

At any point in the plan, the user can inform the system of a predicate that is now true (though SIPE may have thought it was false). The program will look through the plan and find all goals that are affected by this new predicate. Since SIPE understands the rationale of nodes in the plan (through purposes), it can determine how changes affect the plan. For example, if a later purpose is suddenly accomplished unexpectedly, SIPE can notice the helpful effect and eliminate a whole section of the plan since it knows the preparatory steps are only there to accomplish the purpose. If an unexpected occurrence causes a problem, the system will suggest all the solutions it can find. SIPE's repertoire of techniques for finding such solutions is not very sophisticated, however. It includes: (1) instantiating a variable differently (e.g., to use a different resource if something has gone wrong with the one originally used in the plan), (2) finding relevant operators to accomplish a goal that is no longer true (and inserting the new subplan correctly in the original plan), and (3) finding a higher level from which to replan

37

if the problems are widespread.

SIPE's execution monitoring capabilities are an extension of those in previous domain independent planners primarily because the explicit representation of purposes allows more sophisticated replanning. Directions for future research in execution monitoring include using meta-planning to analyze the problem that has occurred, planning for use of information-gathering operations, and using the resulting information to make more sophisticated revisions to plans.

7. Performance of SIPE

SIPE has been tested in four different domains: the blocks world, cooking, aircraft operations, and a simple robotics assembly task. These domains do not have large branching factors or search spaces so the automatic search can find solutions. The cooking domain was encoded to demonstrate the use of resource reasoning. SIPE operators naturally represented requirements for frying pans and burners during the cooking of a dish. Problems such as cooking four dishes with three pans on two burners were handled efficiently by the resource reasoning mechanisms in SIPE. Handling a problem means producing plans for cooking as many dishes as possible in parallel, with enough serialization to get the task accomplished with the available resources. Such plans consisted of dozens of nodes in our simple cooking world.

The standard blocks world was encoded in SIPE, with some enrichments (e.g., more than one block could be on top of another). Use of deductive operators made the PUTON operator more readable. Use of resource reasoning enabled SIPE to quickly find and correct parallel interaction problems. All problems of building three and five block towers can be efficiently solved as long as they do not involve shuffling of actions between two parallel branches (i.e., the problem must be solvable by leaving things in parallel, or placing some number of entire parallel branches sequentially before the others). A number of other problems involving properties

38

of the blocks and quantifiers were also handled elegantly (making use of the constraints in SIPE). For example, the problem of getting *some* red block on top of *some* blue block is easily represented and solved. (SIPE will choose a red block and a blue block that are already clear, if such exist.)

The aircraft operations problem involved planning the launch of a number of airplanes, given their initial locations. This may involve checking for airworthiness, finding paths from a parking space to a fuel station to a runway, and clearing a path if no clear one exists. Problems in this domain included numerous aircraft and long paths, so the plans generated contained hundreds of nodes. Using the automatic search to find paths on the grid would have lead to a combinatorial explosion, so path variables were instantiated to a list of locations by a special purpose generator. SIPE's ability to allow operators to loop over a list enabled the system to generate easily the goals of clearing the locations specified in the list produced by the generator. Using these abilities, SIPE was able automatically to generate large plans for correctly launching many planes from parallel runways.

The robotics problem encoded was similar to a blocks-world problem except that the utilization of the robot arm was planned, and the mating of two parts was represented. (This means the system must deduce that, when an object moves, all things mated to it move with it). SIPE was adequate for representing this problem.

8. Conclusion

SIPE's operator description language was designed to be easy to understand (to enable graceful interaction) while being more powerful than those found in previous domain independent planners. Constraints, resources, and deductive operators all contribute to the power of the representation. Deductive operators allow quantified variables and therefore can be used to make fairly sophisticated deductions, thus eliminating the need to express effects in operators when they can be deduced. They are also useful in distinguishing main effects from

39

side effects.

One of the most important features of SIPE is the ability to constrain the possible values of variables. It is well known that this allows more efficient planning, since choices can be delayed until information has been accumulated. Other advantages of constraints, however, are also critical. A key consideration is that constraints allow convenient expression of a much wider range of problems. Constraint satisfaction efficiently finds variable instantiations by taking advantage of the fact that invariant properties of objects are encoded in the type hierarchy. Constraints also help prevent harmful parallel interactions.

SIPE incorporates several new mechanisms able to assist in dealing with the parallel interaction problem. The most significant of these mechanisms is the ability to reason about resources. It has been beneficial in user interaction to have reasoning about resources as a central part of the system, because resources seem to be a natural and intuitive way to think about objects in many domains. Combined with the system's ability to post constraints, resource reasoning helps the system avoid many harmful interactions, helps it recognize sooner those interactions that do occur, and helps the system solve some of these interactions more quickly. SIPE'S handling of interactions is also improved by its abilities to differentiate side effects and to correctly determine the purposes of actions.

Purposes in SIPE operators are used to coordinate higher level effects with lower-level plans, and to determine the rationale of plan nodes within a hierarchical level. Being able to mention purposes explicitly provides the flexibility needed to represent many domains. Knowing the rationale, SIPE is able to better correct harmful parallel interactions, and to replan more efficiently when things go wrong during execution.

A major difference between SIPE and previous planners is that SIPE is interactive. Its interactive capabilities help the user guide and direct the planning process, allowing alternative plans to be explored concurrently by means of the context mechanism. Thus the user can shift focus as he pleases without being required to understand the program's search strategy or backtracking algorithm. Development of SIPE's automatic planning algorithm has illuminated

why domain independent planners cannot do interesting meta-planning.

REFERENCES

1. Fikes, R., Hart, P., and Nilsson, N., "Learning and Executing Generalized Robot Plans", *Readings in Artificial Intelligence*, Nilsson and Webber, ed., Tioga Publishing, Palo Alto, California, 1981, pp. 231–249.

2. de Kleer et al., "AMORD Explicit control of reasoning", *Proceedings of Symposium on Artificial Intelligence and Programming Languages*, SIGART No. 64, 1977, pp. 116-125.

3. Grosz, B. et al., "TEAM: A Transportable Natural-Language System", Technical Note 263, SRI International Artificial Intelligence Center, Menlo Park, California, 1982.

4. Hayes, P.J., "In Defense of Logic", *Proceedings IJCAI-77*, Cambridge, Massachusetts, 1977, pp. 559-565.

5. Hayes-Roth, B., Hayes-Roth, F., Rosenschein, S., and Cammarata, S., "Modeling Planning as an Incremental, Opportunistic Process", *Proceedings IJCAI-79*, Tokyo, Japan, 1979, pp. 375–383.

6. McDermott, D., "A Temporal Logic for Reasoning About Processes and Plans", Cognitive Science, forthcoming.

7. Robinson, A.E., and Wilkins D.E., "Representing Knowledge in an Interactive Planner" *Proceedings of the First Annual Conference of the AAAI*, Stanford, California, 1980, pp. 148–150.

8. Robinson. A.E., Final Report on the SPOT Project, SRI International Artificial Intelligence Center. Menlo Park, California, forthcoming.

9. Rosenschein, S., "Plan Synthesis: A Logical Perspective", *Proceedings IJCAI-81*, Vancouver, British Columbia, 1981, pp. 331-337.

10. Sacerdoti, E., *A Structure for Plans and Behavior*, Elsevier, North-Holland, New York, 1977.

11. Sridharan, N., and Bresina, J., "Plan Formation in Large, Realistic Domains", *Proceedings CSCSI Conference*, Saskatoon, Saskatchewan, 1982, pp. 12-18.

12. Stefik, M., "Planning with Constraints", Report STAN-CS-80-784, Computer Science Department, Stanford University, 1980.

13. Stefik, M., "Planning and Meta-planning", *Readings in Artificial Intelligence*, Nilsson and Webber, ed., Tioga Publishing, Palo Alto, California, 1981, pp. 272–286.

14. Tate, A., "Generating Project Networks", *Proceedings IJCAI-77*, Cambridge, Massachusetts, 1977, pp. 888–893.

15. van Melle, W., "A Domain-Independent Production-Rule System for Consultation Programs", *Proceedings IJCAI-79*, Tokyo, Japan, 1979, pp. 923–925.

16. Vere, S., "Planning in Time: Windows and Durations for Activities and Goals", Jet Propulsion Lab, Pasadena, California, November 1981.

17. Waldinger, R., "Achieving Several Goals Simultaneously", *Readings in Artificial Intelligence*, Nilsson and Webber, ed., Tioga Publishing, Palo Alto, California, 1981, pp. 250–271.

18. Weyhrauch, R., "Prolegomena To a Theory of Mechanized Formal Reasoning", *Readings in Artificial Intelligence*, Nilsson and Webber, ed., Tioga Publishing, Palo Alto, California, 1981, pp. 173–191.

19. Wilensky, R., "Meta-planning", *Proceedings AAAI-80*, Stanford, California, 1980, pp. 334-336.

20. Wilkins, D., "Using patterns and plans in chess", *Readings in Artificial Intelligence*, Nilsson and Webber, ed., Tioga Publishing, Palo Alto, California, 1981, pp. 390–409.

END

FILMED

3-83

DTIC